Deep Learning (1)

- Multi-layer Neural Network

Yueming Wang

2016. 12. 2

What's deep learning

Deep learning (*deep machine learning*, or *deep structured learning*, or *hierarchical learning*, or sometimes *DL*) is a branch of machine learning based on a set of algorithms that attempt to model high-level abstractions in data by using model architectures, with complex structures or otherwise, composed of multiple non-linear transformations.[1](p198)[2][3][4]

Deep learning is part of a broader family of machine learning methods based on learning representations of data. An observation (e.g., an image) can be represented in many ways such as a vector of intensity values per pixel, or in a more abstract way as a set of edges, regions of particular shape, etc.. Some representations make it easier to learn tasks (e.g., face recognition or facial expression recognition[5]) from examples. One of the promises of deep learning is replacing handcrafted features with efficient algorithms for unsupervised or semi-supervised feature learning and hierarchical feature extraction.[6]

Research in this area attempts to make better representations and create models to learn these representations from large-scale unlabeled data. Some of the representations are inspired by advances in neuroscience and are loosely based on interpretation of information processing and communication patterns in a nervous system, such as neural coding which attempts to define a relationship between the stimulus and the neuronal responses and the relationship among the electrical activity of the neurons in the brain.[7]

Various deep learning architectures such as deep neural networks, convolutional deep neural networks, deep belief networks and recurrent neural networks have been applied to fields like computer vision, automatic speech recognition, natural language processing, audio recognition and bioinformatics where they have been shown to produce state-of-the-art results on various tasks.

What we will discuss today is convolutional deep neural network (CNN)

Success of CNN

Image classification

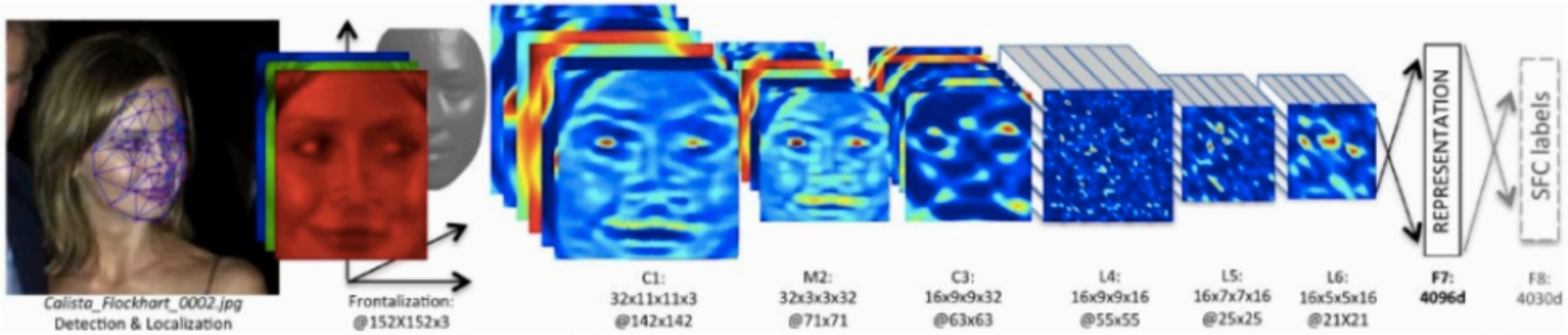
IMAGENET Large Scale Visual Recognition Challenge 2014 (ILSVRC2014)

Team name	Entry description	**Localization error**	Classification error
VGG	a combination of multiple ConvNets (by averaging)	0.253231	0.07405
VGG	a combination of multiple ConvNets (fusion weights learnt on the validation set)	0.253501	0.07407
VGG	a combination of multiple ConvNets, including a net trained on images of different size (fusion done by averaging); detected boxes were not updated	0.255431	0.07337
VGG	a combination of multiple ConvNets, including a net trained on images of different size (fusion weights learnt on the validation set); detected boxes were not updated	0.256167	0.07325
GoogLeNet	Model with localization ~26% top5 val error.	0.264414	0.14828
GoogLeNet	Model with localization ~26% top5 val error, limiting number of classes.	0.264425	0.12724
VGG	a single ConvNet (13 convolutional and 3 fully-connected layers)	0.267184	0.08434
SYSU_Vision	We compared the class-specific localization accuracy of solution 1 and solution 2 by the validation set. Then we chosen better solution on each class based on the accuracy. General speaking, solution 2 outformed solution 1 when there were multiple objects in the image or the objects are relatively small.	0.31899	0.14446
MIL	5 top instances predicted using FV-CNN	0.337414	0.20734
MIL	5 top instances predicted using FV-CNN + class specific window size rejection. Flipped training images are added.	0.33843	0.21023

Success of CNN

Face recognition

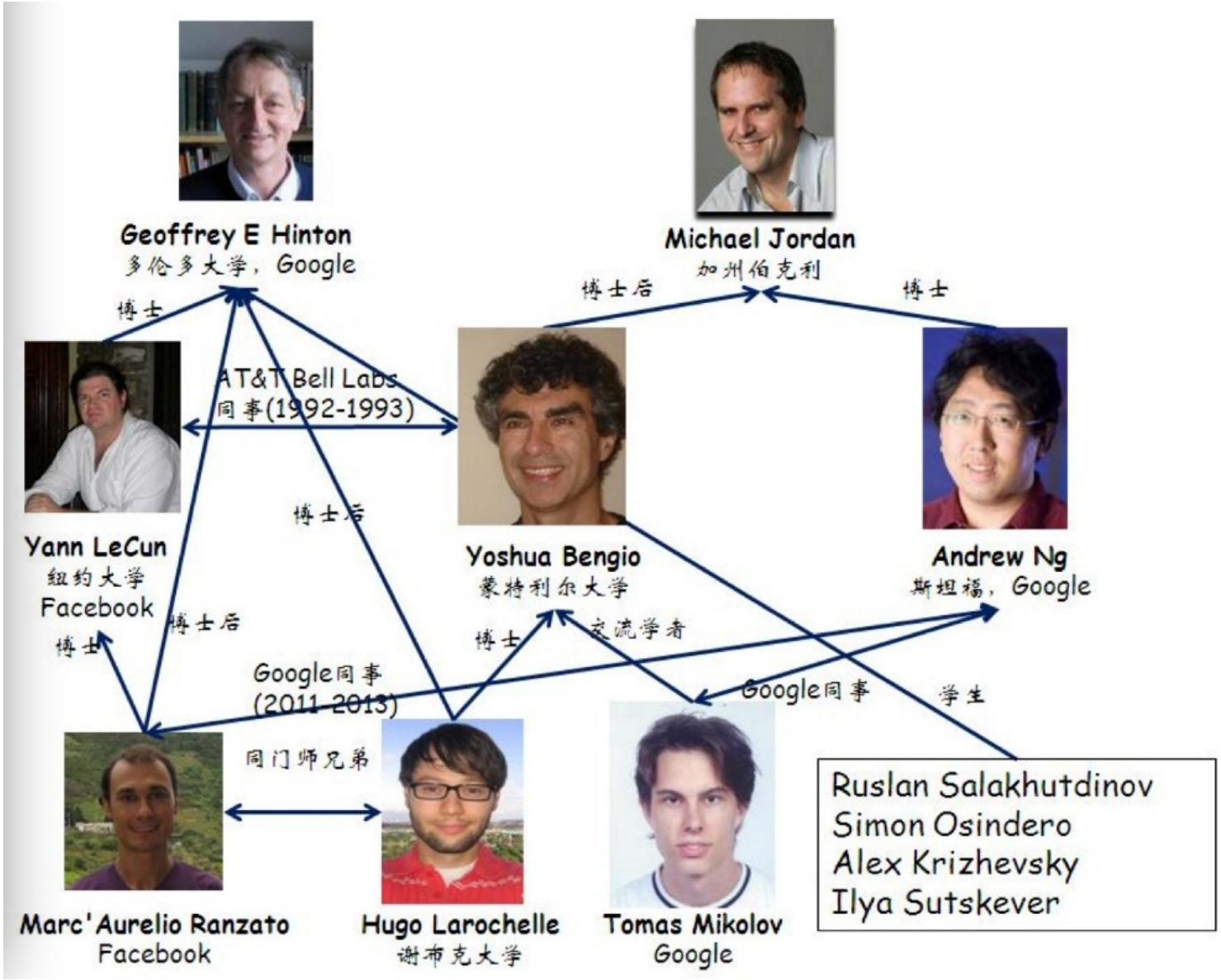
Deepface



simple classifier. Our method reaches an accuracy of 97.35% on the Labeled Faces in the Wild (LFW) dataset, reducing the error of the current state of the art by more than 27%, closely approaching human-level performance.

Taigman, Yaniv, et al. "Deepface: Closing the gap to human-level performance in face verification." Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Conference on. IEEE, 2014.

Who is who in deep learning?



Readings - Deep learning

- Getting started, UFLDL tutorial

  http://ufldl.stanford.edu/wiki/index.php/UFLDL_Tutorial

- LISA deep learning tutorial:

  http://deeplearning.net/tutorial/deeplearning.pdf

- Review paper: Learning Deep Architectures for AI, 2009

- The tutorial on deep learning for vision from cvpr2014:
  https://sites.google.com/site/deeplearningcvpr2014/

- A free online book: neural networks and deep learning by Michael Nielsen:

  http://neuralnetworksanddeeplearning.com/index.html

- Yann LeCun's tutorial (ICML'2013):

  http://www.cs.nyu.edu/~yann/talks/lecun-ranzato-icml2013.pdf

- For an exposition of neural networks in circuits and code, check out Understanding Neural Networks from a Programmer's Perspective by Andrej Karpathy (Stanford)

Readings – Convolutional neural network

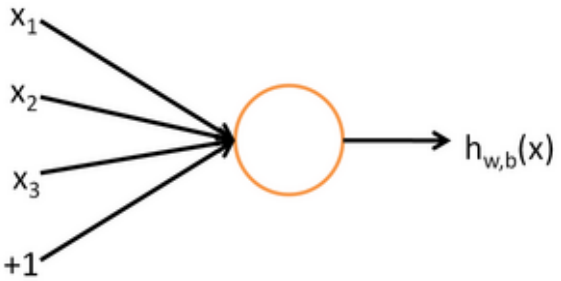
- LeNet-5: Yann LeCun et al. Gradient-Based Learning Applied to Document Recognition, Proceedings of IEEE, 1998. (basic)
- ICCV'09: What is the best multi-stage architecture for object recognition? (Yann LeCun, discuss architecture)
- Nips'12: ImageNet Classification with Deep Convolutional Neural Networks. (Hinton's imageNet work)
- CVPR'10: Learning Mid-Level Features For Recognition (Yann LeCun, feature analysis)
- ICDAR'03: Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis. (How to deal with small number of samples)
- Derivation: Notes on Convolutional Neural Networks.
- Multi-GPU Training of ConvNets (Facebook AI Group)
- A very good overview of these tricks can be found in Efficient BackProp by Yann LeCun: http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf
-

Multi-layer neural network

Neural networks

Consider a supervised learning problem where we have access to labeled training examples $(x^{(i)}, y^{(i)})$. Neural networks give a way of defining a complex, non-linear form of hypotheses $h_{W,b}(x)$, with parameters W, b that we can fit to our data.

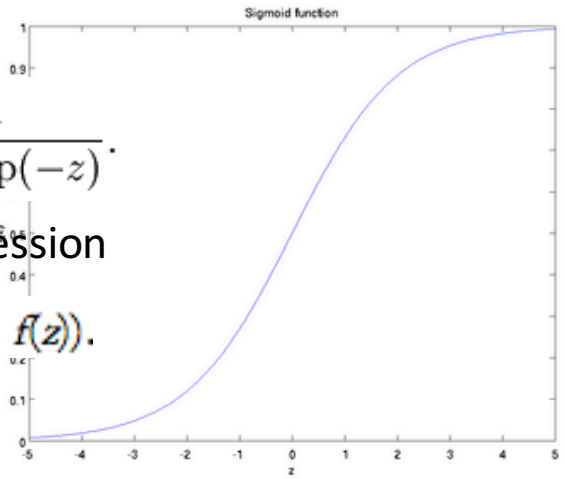
A simplest possible neural network



$h_{W,b}(x) = f(W^T x) = f(\sum_{i=1}^{3} W_i x_i + b)$, where $f : \Re \mapsto \Re$ is called the activation function

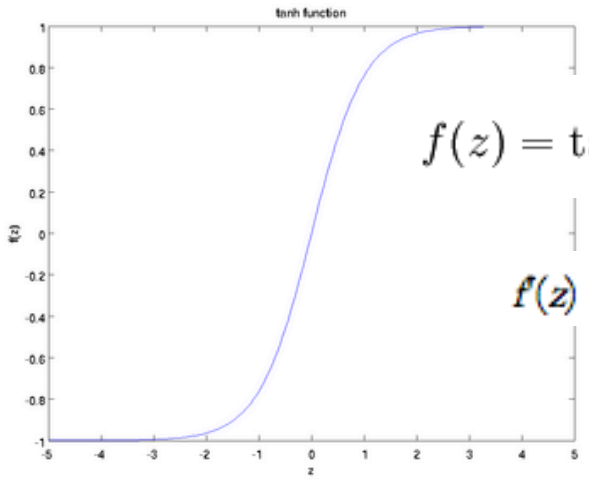
f can be sigmoid function or tanh function

$$f(z) = \frac{1}{1 + \exp(-z)}.$$



logistic regression

$$f'(z) = f(z)(1 - f(z)).$$



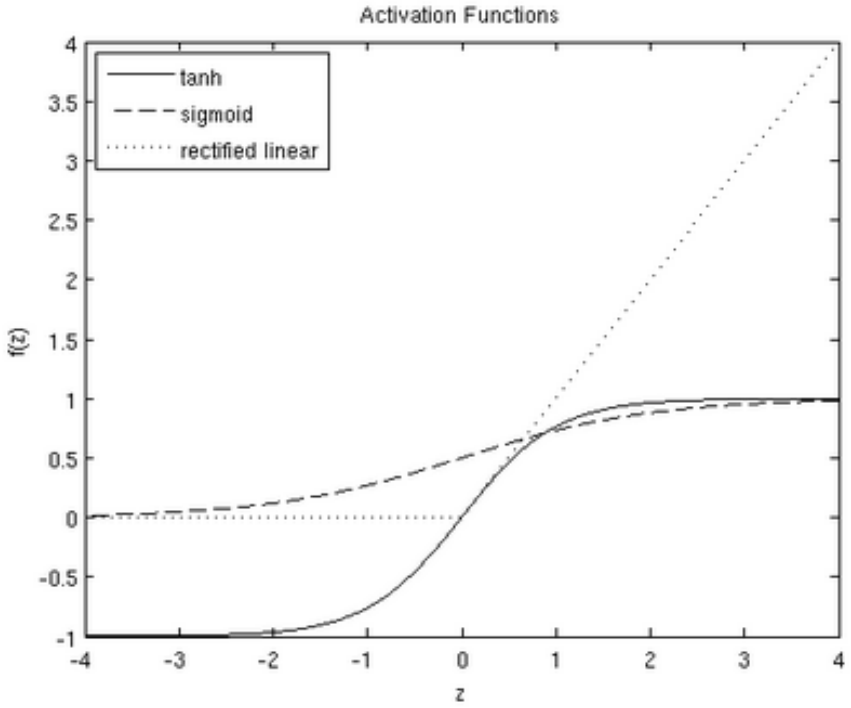
$$f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

$$f'(z) = 1 - (f(z))^2.$$

Neural networks

Recent research has found a different activation function, the rectified linear function, often works better in practice for deep neural networks. This activation function is different from sigmoid and tanh because it is not bounded or continuously differentiable. The rectified linear activation function is given by,

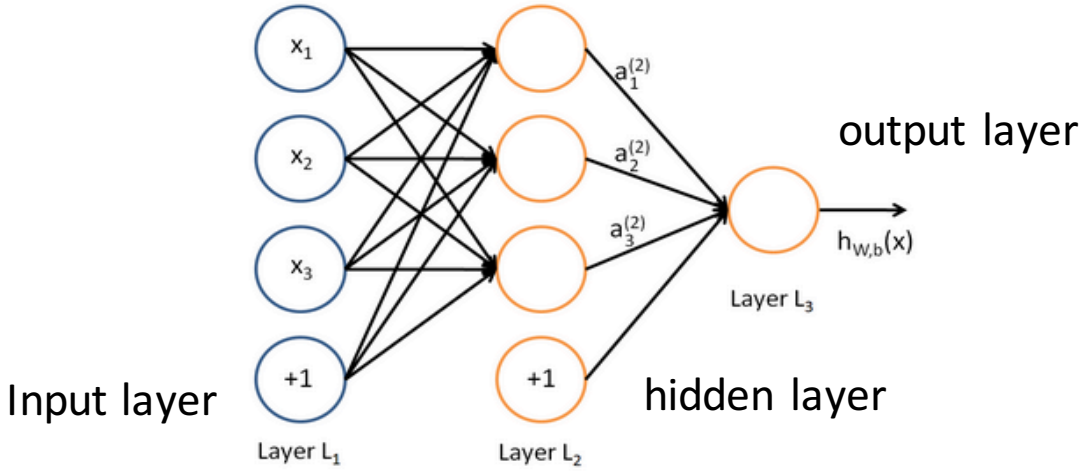
$$f(z) = \max(0, x).$$



The rectified linear function has gradient 0 when $z \leq 0$ and 1 otherwise. The gradient is undefined at $z = 0$, though this doesn't cause problems in practice because we average the gradient over many training examples during optimization.

A small neural network

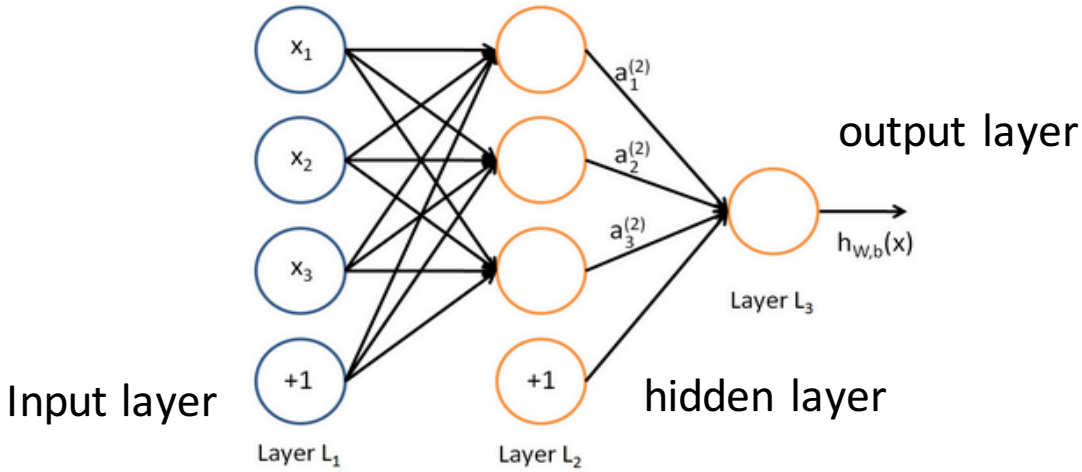
A neural network is put together by hooking together many of our simple "neurons," so that the output of a neuron can be the input of another. For example, here is a small neural network:



Notations:

1. let n_l denote the number of layers in our network; thus $n_l = 3$ in our example
2. label layer l as L_l, so layer L_1 is the input layer, and layer L_{n_l} the output layer.
3. parameters $(W,b) = (W^{(1)},b^{(1)},W^{(2)},b^{(2)})$, where we write $W_{ij}^{(l)}$ to denote the parameter (or weight) associated with the connection between unit j in layer l, and unit i in layer $l+1$. (Note the order
4. $b_i^{(l)}$ is the bias associated with unit i in layer $l+1$. Thus, in our example, we have $W^{(1)} \in \Re^{3\times 3}$, and $W^{(2)} \in \Re^{1\times 3}$.
5. We will write $a_i^{(l)}$ to denote the activation (meaning output value) of unit i in layer l.

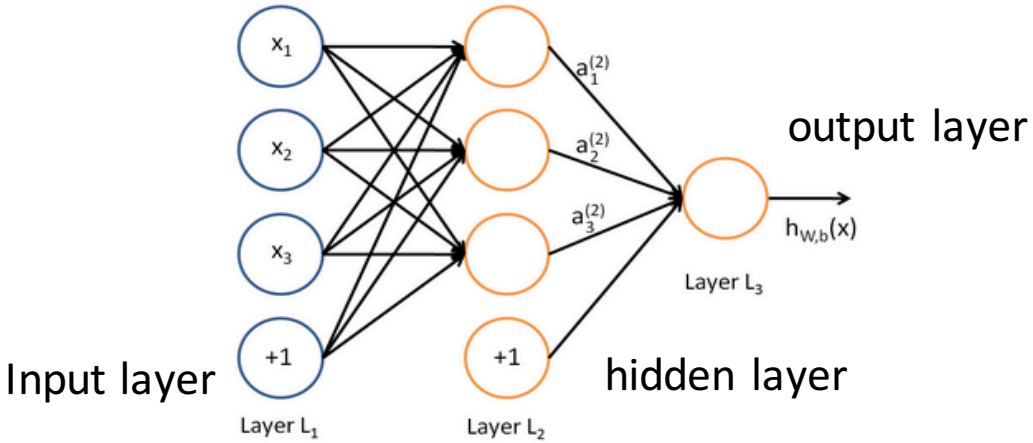
A small neural network



$$a_1^{(2)} = f(W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3 + b_1^{(1)})$$
$$a_2^{(2)} = f(W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3 + b_2^{(1)})$$
$$a_3^{(2)} = f(W_{31}^{(1)}x_1 + W_{32}^{(1)}x_2 + W_{33}^{(1)}x_3 + b_3^{(1)})$$
$$h_{W,b}(x) = a_1^{(3)} = f(W_{11}^{(2)}a_1^{(2)} + W_{12}^{(2)}a_2^{(2)} + W_{13}^{(2)}a_3^{(2)} + b_1^{(2)})$$

In the sequel, we also let $z_i^{(l)}$ denote the total weighted sum of inputs to unit i in layer l, including the bias term (e.g., $z_i^{(2)} = \sum_{j=1}^{n} W_{ij}^{(1)}x_j + b_i^{(1)}$), so that $a_i^{(l)} = f(z_i^{(l)})$.

A small neural network



$$a_1^{(2)} = f(W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3 + b_1^{(1)})$$
$$a_2^{(2)} = f(W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3 + b_2^{(1)})$$
$$a_3^{(2)} = f(W_{31}^{(1)}x_1 + W_{32}^{(1)}x_2 + W_{33}^{(1)}x_3 + b_3^{(1)})$$
$$h_{W,b}(x) = a_1^{(3)} = f(W_{11}^{(2)}a_1^{(2)} + W_{12}^{(2)}a_2^{(2)} + W_{13}^{(2)}a_3^{(2)} + b_1^{(2)})$$

Extend the activation function f(.) to apply to vectors in an element-wise fashion (i.e., f([z1,z2,z3]) = [f(z1),f(z2),f(z3)]), then we can write the equations above more compactly as:

$$z^{(2)} = W^{(1)}x + b^{(1)}$$
$$a^{(2)} = f(z^{(2)})$$
$$z^{(3)} = W^{(2)}a^{(2)} + b^{(2)}$$
$$h_{W,b}(x) = a^{(3)} = f(z^{(3)})$$

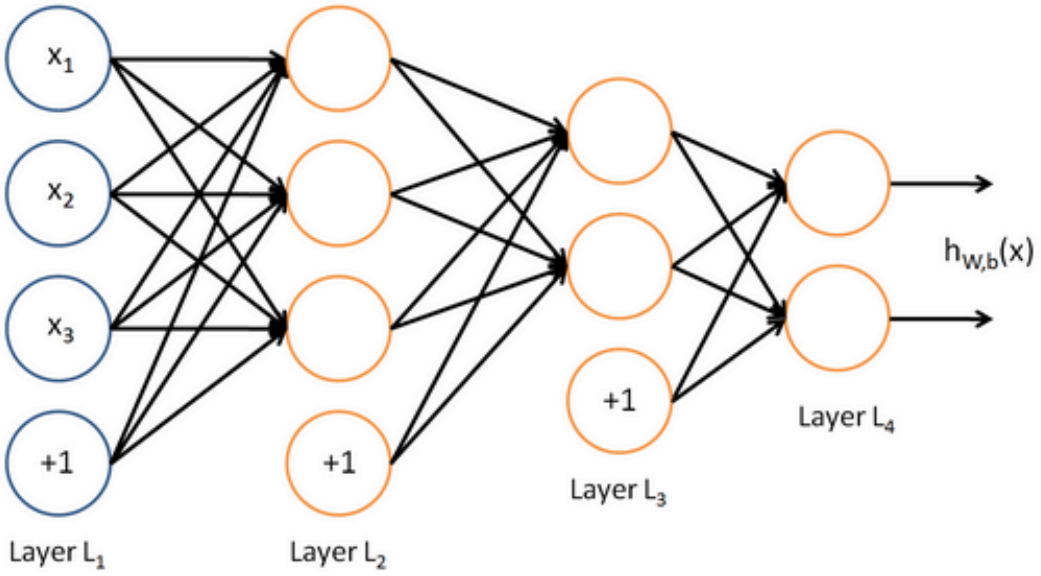
Generally,

$$z^{(l+1)} = W^{(l)}a^{(l)} + b^{(l)}$$
$$a^{(l+1)} = f(z^{(l+1)})$$

We call this step forward propagation.

Neural networks (more outputs if multiple outputs)

Neural networks can also have multiple output units. For example, here is a network with two hidden layers layers L_2 and L_3 and two output units in layer L_4:



a **feedforward** neural network.

how to solve it to obtain optimal W and b?

Backpropagation Algorithm

Loss function

Suppose we have a fixed training set $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ of m training examples. We can train our neural network using batch gradient descent. In detail, for a single training example (x,y), we define the cost function with respect to that single example to be:

$$J(W, b; x, y) = \frac{1}{2} \left\| h_{W,b}(x) - y \right\|^2 .$$

This is a (one-half) squared-error cost function. Given a training set of m examples, we then define the overall cost function to be:

$$\begin{aligned} J(W, b) &= \left[\frac{1}{m} \sum_{i=1}^{m} J(W, b; x^{(i)}, y^{(i)}) \right] + \frac{\lambda}{2} \sum_{l=1}^{n_l - 1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} \left(W_{ji}^{(l)} \right)^2 \\ &= \left[\frac{1}{m} \sum_{i=1}^{m} \left(\frac{1}{2} \left\| h_{W,b}(x^{(i)}) - y^{(i)} \right\|^2 \right) \right] + \frac{\lambda}{2} \sum_{l=1}^{n_l - 1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} \left(W_{ji}^{(l)} \right)^2 \end{aligned}$$

The first term in the definition of $J(W,b)$ is an average sum-of-squares error term. The second term is a regularization term (also called a weight decay term) that tends to decrease the magnitude of the weights, and helps prevent overfitting.

For classification, we let y = 0 or 1 represent the two class labels (recall that the sigmoid activation function outputs values in [0,1]; if we were using a tanh activation function, we would instead use -1 and +1 to denote the labels). For regression problems, we first scale our outputs to ensure that they lie in the [0,1] range (or if we were using a tanh activation function, then the [− 1,1] range).

Optimization – gradient descent

Our goal is to minimize $J(W,b)$ as a function of W and b. To train our neural network, we will initialize each parameter $W_{ij}^{(l)}$ and each $b_i^{(l)}$ to a small random value near zero (say according to a $Normal(0,\varepsilon^2)$ distribution for some small ε, say 0.01), and then apply an optimization algorithm such as batch gradient descent

$$J(W,b) = \left[\frac{1}{m}\sum_{i=1}^{m} J(W,b;x^{(i)},y^{(i)})\right] + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(W_{ji}^{(l)}\right)^2$$

$$= \left[\frac{1}{m}\sum_{i=1}^{m}\left(\frac{1}{2}\left\|h_{W,b}(x^{(i)}) - y^{(i)}\right\|^2\right)\right] + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(W_{ji}^{(l)}\right)^2$$

$$J(W,b;x,y) = \frac{1}{2}\left\|h_{W,b}(x) - y\right\|^2.$$

we will initialize each parameter $W_{ij}^{(l)}$ and each $b_i^{(l)}$ to a small random value near zero (say according to a $Normal(0,\varepsilon^2)$ distribution for some small ε, say 0.01), and then apply an optimization algorithm such as batch gradient descent.

$$W_{ij}^{(l)} = W_{ij}^{(l)} - \alpha\frac{\partial}{\partial W_{ij}^{(l)}}J(W,b)$$

$$b_i^{(l)} = b_i^{(l)} - \alpha\frac{\partial}{\partial b_i^{(l)}}J(W,b)$$

Gradient

Objective function:

$$J(W,b) = \left[\frac{1}{m}\sum_{i=1}^{m} J(W,b;x^{(i)},y^{(i)})\right] + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(W_{ji}^{(l)}\right)^2$$

$$= \left[\frac{1}{m}\sum_{i=1}^{m}\left(\frac{1}{2}\left\|h_{W,b}(x^{(i)}) - y^{(i)}\right\|^2\right)\right] + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(W_{ji}^{(l)}\right)^2$$

$$J(W,b;x,y) = \frac{1}{2}\left\|h_{W,b}(x) - y\right\|^2.$$

Backpropagation: 反向传播算法

We will first describe how backpropagation can be used to compute $\frac{\partial}{\partial W_{ij}^{(l)}} J(W,b;x,y)$ and $\frac{\partial}{\partial b_i^{(l)}} J(W,b;x,y)$, the partial derivatives of the cost function $J(W,b;x,y)$ defined with respect to a single example (x,y). Once we can compute these, we see that the derivative of the overall cost function $J(W,b)$ can be computed as:

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(W,b) = \left[\frac{1}{m}\sum_{i=1}^{m}\frac{\partial}{\partial W_{ij}^{(l)}} J(W,b;x^{(i)},y^{(i)})\right] + \lambda W_{ij}^{(l)}$$

$$\frac{\partial}{\partial b_i^{(l)}} J(W,b) = \frac{1}{m}\sum_{i=1}^{m}\frac{\partial}{\partial b_i^{(l)}} J(W,b;x^{(i)},y^{(i)})$$

Backpropagation for a single example

For a single example:

$$J(W,b;x,y) = \frac{1}{2}\left\|h_{W,b}(x) - y\right\|^2.$$

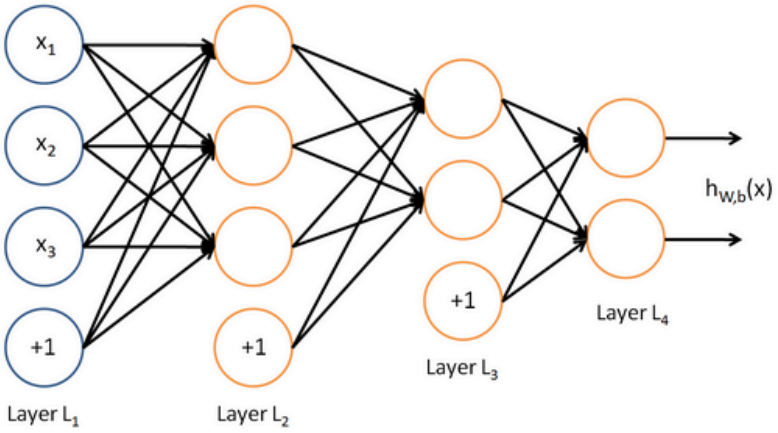
For each layer l :

$$a_i^{(l)} = f(z_i^{(l)})$$

$$z_i^{(l)} = \sum_{j=1}^{S_{l-1}} W_{ij}^{(l-1)} a_j^{(l-1)} + b_j^{(l-1)}$$

For output layer n_l :

$$h_{w,b}(x)_i = a_i^{(n_l)}$$



Idea of BP:

- Given an example (x, y) and the initialized W and b, compute the activations through the network;
- From the output layer, compute the partial derivative of J about $z_i^{(nl)}$, called $delta_i^{(nl)}$;
- Layer by Layer, from n_l to 1, compute the partial derivative of output about $z_i^{(l)}$, i.e., $delta_i^{(l)}$, which has a simple relation with $delta_i^{(l+1)}$;
- The partial derivative of J about $W_{ij}^{(l)}$ can be easily derived from $delta_i^{(l+1)}$.

Backpropagation for a single example

For a single example:

$$J(W,b;x,y) = \frac{1}{2}\left\|h_{W,b}(x) - y\right\|^2 .$$

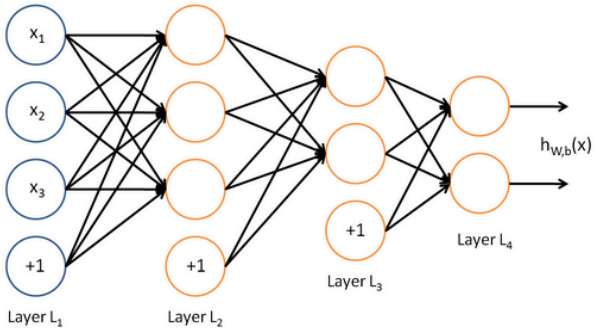
For each layer l :

$$a_i^{(l)} = f(z_i^{(l)})$$

$$z_i^{(l)} = \sum_{j=1}^{S_{l-1}} W_{ij}^{(l-1)} a_j^{(l-1)} + b_j^{(l-1)}$$

For output layer n_l :

$$h_{w,b}(x)_i = a_i^{(n_l)}$$



1. Perform a feedforward pass, computing the activations for layers L_2, L_3, and so on up to the output layer L_{n_l}.
2. For each output unit i in layer n_l (the output layer), set

$$\begin{aligned}\delta_i^{(n_l)} &= \frac{\partial}{\partial z_i^{n_l}} J(W,b;x,y) = \frac{\partial}{\partial z_i^{n_l}} \frac{1}{2}\left\|y - h_{W,b}(x)\right\|^2 \\ &= \frac{\partial}{\partial z_i^{n_l}} \frac{1}{2}\sum_{j=1}^{S_{n_l}} (y_j - a_j^{(n_l)})^2 = \frac{\partial}{\partial z_i^{n_l}} \frac{1}{2}\sum_{j=1}^{S_{n_l}} (y_j - f(z_j^{(n_l)}))^2 \\ &= -(y_i - f(z_i^{(n_l)})) \cdot f'(z_i^{(n_l)}) = -(y_i - a_i^{(n_l)}) \cdot f'(z_i^{(n_l)})\end{aligned}$$

3. For $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$

   For each node i in layer l, set

$$\begin{aligned}\delta_i^{(n_l-1)} &= \frac{\partial}{\partial z_i^{n_l-1}} J(W,b;x,y) = \frac{\partial}{\partial z_i^{n_l-1}} \frac{1}{2}\left\|y - h_{W,b}(x)\right\|^2 = \frac{\partial}{\partial z_i^{n_l-1}} \frac{1}{2}\sum_{j=1}^{S_{n_l}} (y_j - a_j^{(n_l)})^2 \\ &= \frac{1}{2}\sum_{j=1}^{S_{n_l}} \frac{\partial}{\partial z_i^{n_l-1}} (y_j - a_j^{(n_l)})^2 = \frac{1}{2}\sum_{j=1}^{S_{n_l}} \frac{\partial}{\partial z_i^{n_l-1}} (y_j - f(z_j^{(n_l)}))^2 \\ &= \sum_{j=1}^{S_{n_l}} -(y_j - f(z_j^{(n_l)})) \cdot \frac{\partial}{\partial z_i^{(n_l-1)}} f(z_j^{(n_l)}) = \sum_{j=1}^{S_{n_l}} -(y_j - f(z_j^{(n_l)})) \cdot f'(z_j^{(n_l)}) \cdot \frac{\partial z_j^{(n_l)}}{\partial z_i^{(n_l-1)}} \\ &= \sum_{j=1}^{S_{n_l}} \delta_j^{(n_l)} \cdot \frac{\partial z_j^{(n_l)}}{\partial z_i^{n_l-1}} = \sum_{j=1}^{S_{n_l}} \left(\delta_j^{(n_l)} \cdot \frac{\partial}{\partial z_i^{n_l-1}} \sum_{k=1}^{S_{n_l-1}} f(z_k^{n_l-1}) \cdot W_{jk}^{n_l-1} \right) \\ &= \sum_{j=1}^{S_{n_l}} \delta_j^{(n_l)} \cdot W_{ji}^{n_l-1} \cdot f'(z_i^{n_l-1}) = \left(\sum_{j=1}^{S_{n_l}} W_{ji}^{n_l-1} \delta_j^{(n_l)} \right) f'(z_i^{n_l-1})\end{aligned}$$

Replacing n_l and n_{l-1} with l+1 and l, we get:

$$\delta_i^{(l)} = \left(\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)} \right) f'(z_i^{(l)})$$

Backpropagation for a single example

1. Perform a feedforward pass, computing the activations for layers L_2, L_3, and so on up to the output layer L_{n_l}.
2. For each output unit i in layer n_l (the output layer), set

$$\delta_i^{(n_l)} = \frac{\partial}{\partial z_i^{(n_l)}} \frac{1}{2} \|y - h_{W,b}(x)\|^2 = -(y_i - a_i^{(n_l)}) \cdot f'(z_i^{(n_l)})$$

3. For $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$

   For each node i in layer l, set

$$\delta_i^{(l)} = \left(\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)} \right) f'(z_i^{(l)})$$

4. Compute the desired partial derivatives, which are given as:

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(W, b; x, y) = a_j^{(l)} \delta_i^{(l+1)}$$

$$\frac{\partial}{\partial b_i^{(l)}} J(W, b; x, y) = \delta_i^{(l+1)}.$$

Backpropagation for a single example (matrix-vectorial)

We will use "•" to denote the element-wise product operator, so that if $a = b \bullet c$, then $a_i = b_i c_i$.

we also do the same for $f'(\cdot)$ (so that $f'([z_1, z_2, z_3]) = [f'(z_1), f'(z_2), f'(z_3)]$).

The algorithm can then be written:

1. Perform a feedforward pass, computing the activations for layers L_2, L_3, up to the output layer L_{n_l}, using the equations defining the forward propagation steps
2. For the output layer (layer n_l), set
$$\delta^{(n_l)} = -(y - a^{(n_l)}) \bullet f'(z^{(n_l)})$$
3. For $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$
Set
$$\delta^{(l)} = \left((W^{(l)})^T \delta^{(l+1)}\right) \bullet f'(z^{(l)})$$
4. Compute the desired partial derivatives:
$$\nabla_{W^{(l)}} J(W, b; x, y) = \delta^{(l+1)} (a^{(l)})^T,$$
$$\nabla_{b^{(l)}} J(W, b; x, y) = \delta^{(l+1)}.$$

Implementation note: In steps 2 and 3 above, we need to compute $f'(z_i^{(l)})$ for each value of i. Assuming $f(z)$ is the sigmoid activation function, we would already have $a_i^{(l)}$ stored away from the forward pass through the network. Thus, using the expression that we worked out earlier for $f'(z)$, we can compute this as $f'(z_i^{(l)}) = a_i^{(l)}(1 - a_i^{(l)})$.

Pseudo code of backpropagation

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(W,b) = \left[\frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial W_{ij}^{(l)}} J(W,b;x^{(i)},y^{(i)}) \right] + \lambda W_{ij}^{(l)}$$

$$\frac{\partial}{\partial b_{i}^{(l)}} J(W,b) = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial b_{i}^{(l)}} J(W,b;x^{(i)},y^{(i)})$$

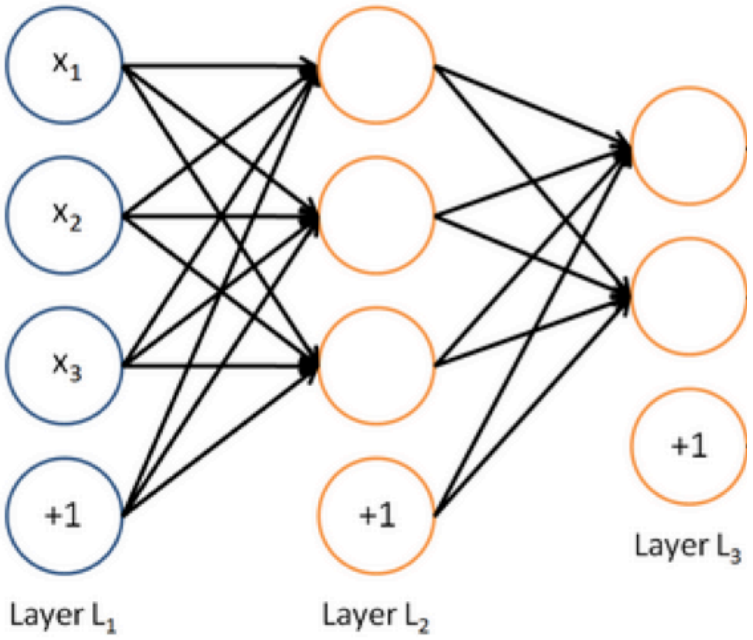
1. Set $\Delta W^{(l)} := 0$, $\Delta b^{(l)} := 0$ (matrix/vector of zeros) for all l.
2. For $i = 1$ to m,
   a. Use backpropagation to compute $\nabla_{W^{(l)}} J(W,b;x,y)$ and $\nabla_{b^{(l)}} J(W,b;x,y)$.
   b. Set $\Delta W^{(l)} := \Delta W^{(l)} + \nabla_{W^{(l)}} J(W,b;x,y)$.
   c. Set $\Delta b^{(l)} := \Delta b^{(l)} + \nabla_{b^{(l)}} J(W,b;x,y)$.
3. Update the parameters:

$$W^{(l)} = W^{(l)} - \alpha \left[\left(\frac{1}{m} \Delta W^{(l)} \right) + \lambda W^{(l)} \right]$$

$$b^{(l)} = b^{(l)} - \alpha \left[\frac{1}{m} \Delta b^{(l)} \right]$$

Exercise of multi-layer neural network

- Dataset: MNIST
- Number of hidden layers: 1, number of nodes in hidden layer: 500
- Activation function in the hidden layer: tanh: $f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$ $f'(z) = 1 - (f(z))^2$.
- Output layer: logistic regression (softmax)
- Loss function: negative Log-likelihood
- Use L2 regularization
- Do NOT use Theano



Exercise: negative log-likelihood

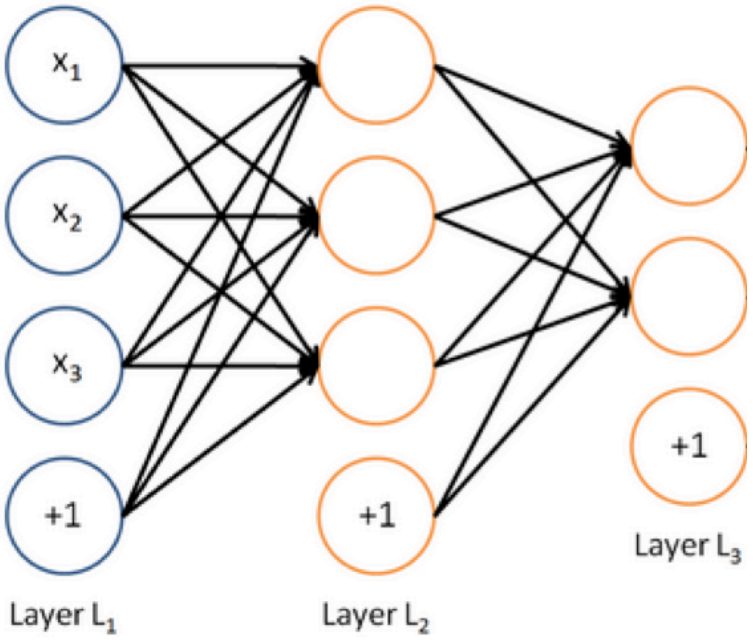
$$J(W,b) = -\left[\frac{1}{m}\sum_{i=1}^{m} J(W,b;x^{(i)},y^{(i)})\right] + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(W_{ji}^{(l)}\right)^2$$

$$J(W,b;x,y) = \sum_{i=1}^{S_{n_l}} 1(y=i)\ln p(y=i|x;W,b)$$

$$p(y=i|x;W,b) = \frac{e^{z_i^{(n_l)}}}{\sum_{l=1}^{S_{n_l}} e^{z_l^{(n_l)}}} = \frac{e^{W_i^{(n_l-1)}a^{(n_l-1)}+b_i^{(n_l-1)}}}{\sum_{l=1}^{S_{n_l}} e^{W_l^{(n_l-1)}a^{(n_l-1)}+b_l^{(n_l-1)}}}$$

$$\frac{\partial}{\partial W_{ij}^{(l)}}J(W,b) = -\left[\frac{1}{m}\sum_{i=1}^{m}\frac{\partial}{\partial W_{ij}^{(l)}}J(W,b;x^{(i)},y^{(i)})\right] + \lambda W_{ij}^{(l)}$$

$$\frac{\partial}{\partial b_i^{(l)}}J(W,b) = -\frac{1}{m}\sum_{i=1}^{m}\frac{\partial}{\partial b_i^{(l)}}J(W,b;x^{(i)},y^{(i)})$$



28x28 500 10

Backpropagation for a single example

$$J(W,b;x,y) = \sum_{i=1}^{S_{n_l}} 1(y=i)\ln p(y=i|x;W,b)$$

log-likelihood

$$p(y=i|x;W,b) = e^{z_i^{(n_l)}} / \sum_{l=1}^{S^{n_l}} e^{z_l^{(n_l)}} = e^{W_i^{(n_l-1)}a^{(n_l-1)}+b_i^{(n_l-1)}} / \sum_{l=1}^{S^{n_l}} e^{W_l^{(n_l-1)}a^{(n_l-1)}+b_l^{(n_l-1)}}$$

$$\delta_i^{(n_l)} = \frac{\partial J(W,b;x,y)}{\partial z_i^{(n_l)}} = \partial \sum_{j=1}^{S_{n_l}} [1(y=j)(\ln e^{z_j^{(n_l)}} - \ln \sum_{l=1}^{S_{n_l}} e^{z_l^{(n_l)}})] / \partial z_i^{(n_l)}$$

$$= \frac{\partial \sum_{j=1}^{S_{n_l}} [1(y=j) z_j^{(n_l)}]}{\partial z_i^{(n_l)}} - \frac{\ln \sum_{l=1}^{S_{n_l}} e^{z_l^{(n_l)}}}{\partial z_i^{(n_l)}}$$

$$= 1(y=i) - \frac{e^{z_i^{(n_l)}}}{\Sigma} = 1(y=i) - p(y=i)$$

Backpropagation for a single example

$$\delta_i^{(n_l)} = 1(y=i) - p(y=i)$$

$$\delta_i^{(n_l-1)} = \frac{\partial J(W,b;x,y)}{\partial z_i^{(n_l-1)}} = \frac{\partial \sum_{j=1}^{S_{n_l}} [1(y=j) z_j^{(n_l)}]}{\partial z_i^{(n_l-1)}} - \frac{\ln \sum_{l=1}^{S_{n_l}} e^{z_l^{(n_l)}}}{\partial z_i^{(n_l-1)}}$$

$$= \frac{\partial \sum_{j=1}^{S_{n_l}} [1(y=j)(\sum_{m=1}^{S_{n_l-1}} W_{jm}^{(n_l-1)} f(z_m^{(n_l-1)}) + b_m^{(n_l-1)})]}{\partial z_i^{(n_l-1)}} - \frac{1}{\Sigma} \frac{\partial \sum_{l=1}^{S_{n_l}} e^{z_l^{(n_l)}} (\sum_{m=1}^{S_{n_l-1}} W_{lm}^{(n_l-1)} f(z_m^{(n_l-1)}) + b_m^{(n_l-1)})}{\partial z_i^{(n_l-1)}}$$

$$= \sum_{j=1}^{S_{n_l}} [1(y=j)(W_{ji}^{(n_l-1)} f'(z_i^{(n_l-1)})] - \sum_{l=1}^{S_{n_l}} [p(y=l)(W_{li}^{(n_l-1)} f'(z_i^{(n_l-1)})]$$

$$= [\sum_{l=1}^{S_{n_l}} (1(y=j) - p(y=j)) W_{ji}^{(n_l-1)}] f'(z_i^{(n_l-1)}) = \sum_{l=1}^{S_{n_l}} \delta_j^{(n_l)} W_{ji}^{(n_l-1)}] f'(z_i^{(n_l-1)})$$

So,

$$\delta_i^{(l)} = \left(\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)} \right) f'(z_i^{(l)}) \qquad f(z) = 1 - (f(z))^2.$$

Pseudo code

The previous pseudo code of BP still works, except that the delta of the output layer and the activation becomes tanh.

The algorithm can then be written:

1. Perform a feedforward pass, computing the activations for layers L_2, L_3, up to the output layer L_{n_l}, using the equations defining the forward propagation steps
2. For the output layer (layer n_l), set
$$\delta^{(n_l)} = 1(y = [0,1,2,\ldots,9]^T) - p(y = [0,1,2,\ldots,9]^T)$$
3. For $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$

   Set
$$\delta^{(l)} = \left((W^{(l)})^T \delta^{(l+1)}\right) \bullet f'(z^{(l)})$$
4. Compute the desired partial derivatives:
$$\nabla_{W^{(l)}} J(W, b; x, y) = \delta^{(l+1)} (a^{(l)})^T,$$
$$\nabla_{b^{(l)}} J(W, b; x, y) = \delta^{(l+1)}.$$

$$f'(z) = 1 - (f(z))^2.$$

$$p(y = i \mid x; W, b) = e^{z_i^{(n_l)}} \Big/ \sum_{l=1}^{S^{n_l}} e^{z_l^{(n_l)}}$$

Update parameters

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(W,b) = -\left[\frac{1}{m}\sum_{i=1}^{m} \frac{\partial}{\partial W_{ij}^{(l)}} J(W,b;x^{(i)},y^{(i)})\right] + \lambda W_{ij}^{(l)}$$

$$\frac{\partial}{\partial b_{i}^{(l)}} J(W,b) = -\frac{1}{m}\sum_{i=1}^{m} \frac{\partial}{\partial b_{i}^{(l)}} J(W,b;x^{(i)},y^{(i)})$$

1. Set $\Delta W^{(l)} := 0$, $\Delta b^{(l)} := 0$ (matrix/vector of zeros) for all l.
2. For $i = 1$ to m,
   a. Use backpropagation to compute $\nabla_{W^{(l)}} J(W,b;x,y)$ and $\nabla_{b^{(l)}} J(W,b;x,y)$.
   b. Set $\Delta W^{(l)} := \Delta W^{(l)} - \nabla_{W^{(l)}} J(W,b;x,y)$.
   c. Set $\Delta b^{(l)} := \Delta b^{(l)} - \nabla_{b^{(l)}} J(W,b;x,y)$.
3. Update the parameters:

$$W^{(l)} = W^{(l)} - \alpha\left[\left(\frac{1}{m}\Delta W^{(l)}\right) + \lambda W^{(l)}\right]$$

$$b^{(l)} = b^{(l)} - \alpha\left[\frac{1}{m}\Delta b^{(l)}\right]$$

Code

```python
class MLP(object):

    def __init__(self, n_in=28*28, n_list_hidden_nodes=[500], n_out=10):...

    def feedforward(self, x):

        xx = x
        for each in self.hidden_layer_list:
            each.forward_compute_z_a(xx)
            xx = each.a

        self.output_layer.forward_compute_p_y_given_x(xx)

    def backpropagation(self, x, y, learning_rate, L2_reg):

        self.output_layer.back_compute_delta(y)
        xx = self.hidden_layer_list[-1].a
        self.output_layer.back_update_w_b(xx, learning_rate, L2_reg)
        next_W = self.output_layer.W
        next_delta = self.output_layer.delta
        i = len(self.hidden_layer_list)
        while i > 0:
            curr_hidden_lay = self.hidden_layer_list[i-1]
            curr_hidden_lay.back_delta(next_W, next_delta)

            if i > 1:
                xx = self.hidden_layer_list[i-2].a
            else:
                xx = x
            curr_hidden_lay.back_update_W_b(xx, learning_rat

            next_W = curr_hidden_lay.W
            next_delta = curr_hidden_lay.delta
            i -= 1
```

```python
class hidden_layer(object):

    def __init__(self, n_in, n_out):...


    def forward_compute_z_a(self, x):

        self.z = numpy.dot(x, self.W) + self.b
        self.a = numpy.tanh(self.z) # samples x n_out
        return self.a

    def back_delta(self, next_W, next_delta):

        tt = numpy.dot(next_delta, next_W.transpose()) # samples x n_out
        self.delta = tt * (1 - self.a ** 2)

    def back_update_W_b(self, x, learning_rate, L2_reg):

        # b = timeit.default_timer()
        delta_W = -1.0 * numpy.dot(x.transpose(), self.delta) / x.shape[0]
        # e = timeit.default_timer()
        # print(e - b)
        # exit()

        delta_b = -1.0 * numpy.mean(self.delta, axis=0)

        self.W -= learning_rate * (L2_reg * self.W + delta_W)
        self.b -= learning_rate * delta_b
```

```python
# softmax
class output_layer(object):

    def __init__(self, n_in, n_out):...

    def forward_compute_p_y_given_x(self, x):

        self.exp_x_multiply_W_plus_b = numpy.exp(numpy.dot(x, self.W) + self.b)
        sigma = numpy.sum(self.exp_x_multiply_W_plus_b, axis=1)
        self.p_y_given_x = self.exp_x_multiply_W_plus_b / sigma.reshape(sigma.shape[0], 1)

    def back_compute_delta(self, y):

        yy = numpy.zeros((y.shape[0], self.n_out))
        yy[numpy.arange(y.shape[0]), y] = 1.0
        self.delta = yy - self.p_y_given_x

    def back_update_w_b(self, x, learning_rate, L2_reg):

        delta_W = -1.0 * numpy.dot(x.transpose(), self.delta)/x.shape[0]
        delta_b = -1.0 * numpy.mean(self.delta, axis=0)
        self.W -= learning_rate * (L2_reg * self.W + delta_W)
        self.b -= learning_rate * delta_b
```

Results on MNIST

```
epoch 25, minibatch 2500/2500, validation error 3.280000 %
time elapsed for one epoch 17.330668 s
epoch 26, minibatch 2500/2500, validation error 3.220000 %
time elapsed for one epoch 18.767549 s
epoch 27, minibatch 2500/2500, validation error 3.180000 %
time elapsed for one epoch 18.369037 s
epoch 28, minibatch 2500/2500, validation error 3.180000 %
time elapsed for one epoch 18.665179 s
epoch 29, minibatch 2500/2500, validation error 3.120000 %
```

```
epoch 261, minibatch 2500/2500, validation error 1.900000 %
time elapsed for one epoch 19.149027 s
epoch 262, minibatch 2500/2500, validation error 1.900000 %
time elapsed for one epoch 19.613950 s
epoch 263, minibatch 2500/2500, validation error 1.890000 %
time elapsed for one epoch 19.046455 s
epoch 264, minibatch 2500/2500, validation error 1.890000 %
time elapsed for one epoch 18.668984 s
epoch 265, minibatch 2500/2500, validation error 1.890000 %
time elapsed for one epoch 19.753155 s
```

Considerations in practice

Strictly speaking, finding an optimal set of values for the parameters is not a feasible problem. The good news is that over the last 25 years, researchers have devised various rules of thumb for choosing hyper-parameters in a neural network. A very good overview of these tricks can be found in Efficient BackProp: http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf

Nonlinearity ¶

Two of the most common ones are the $sigmoid$ and the $tanh$ function. For reasons explained in Section 4.4, nonlinearities that are symmetric around the origin are preferred because they tend to produce zero-mean inputs to the next layer (which is a desirable property). Empirically, we have observed that the $tanh$ has better convergence properties.

Weight initialization http://deeplearning.net/tutorial/references.html#xavier10

At initialization we want the weights to be small enough around the origin so that the activation function operates in its linear regime, where gradients are the largest. Other desirable properties, especially for deep networks, are to conserve variance of the activation as well as variance of back-propagated gradients from layer to layer. This allows information to flow well upward and downward in the network and reduces discrepancies between layers. Under some assumptions, a compromise between these two constraints leads to the following initialization: $uniform[-\frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}}, \frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}}]$ for tanh and $uniform[-4*\frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}}, 4*\frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}}]$ for sigmoid. Where fan_{in} is the number of inputs and fan_{out} the number of hidden units. For mathematical considerations please refer to [Xavier10].

Considerations in practice

Learning rate

There is a great deal of literature on choosing a good learning rate. The simplest solution is to simply have a constant rate. Rule of thumb: try several log-spaced values ($10^{-1}, 10^{-2}, \ldots$) and narrow the (logarithmic) grid search to the region where you obtain the lowest validation error.

Decreasing the learning rate over time is sometimes a good idea. One simple rule for doing that is $\frac{\mu_0}{1+d\times t}$ where μ_0 is the initial rate (chosen, perhaps, using the grid search technique explained above), d is a so-called "decrease constant" which controls the rate at which the learning rate decreases (typically, a smaller positive number, 10^{-3} and smaller) and t is the epoch/stage.

Number of hidden units

This hyper-parameter is very much dataset-dependent. Vaguely speaking, the more complicated the input distribution is, the more capacity the network will require to model it, and so the larger the number of hidden units that will be needed (note that the number of weights in a layer, perhaps a more direct measure of capacity, is $D \times D_h$ (recall D is the number of inputs and D_h is the number of hidden units).

Unless we employ some regularization scheme (early stopping or L1/L2 penalties), a typical number of hidden units vs. generalization performance graph will be U-shaped.

Regularization parameter ¶

Typical values to try for the L1/L2 regularization parameter λ are $10^{-2}, 10^{-3}, \ldots$.

Practice multi-layer neural network on the basketball problem

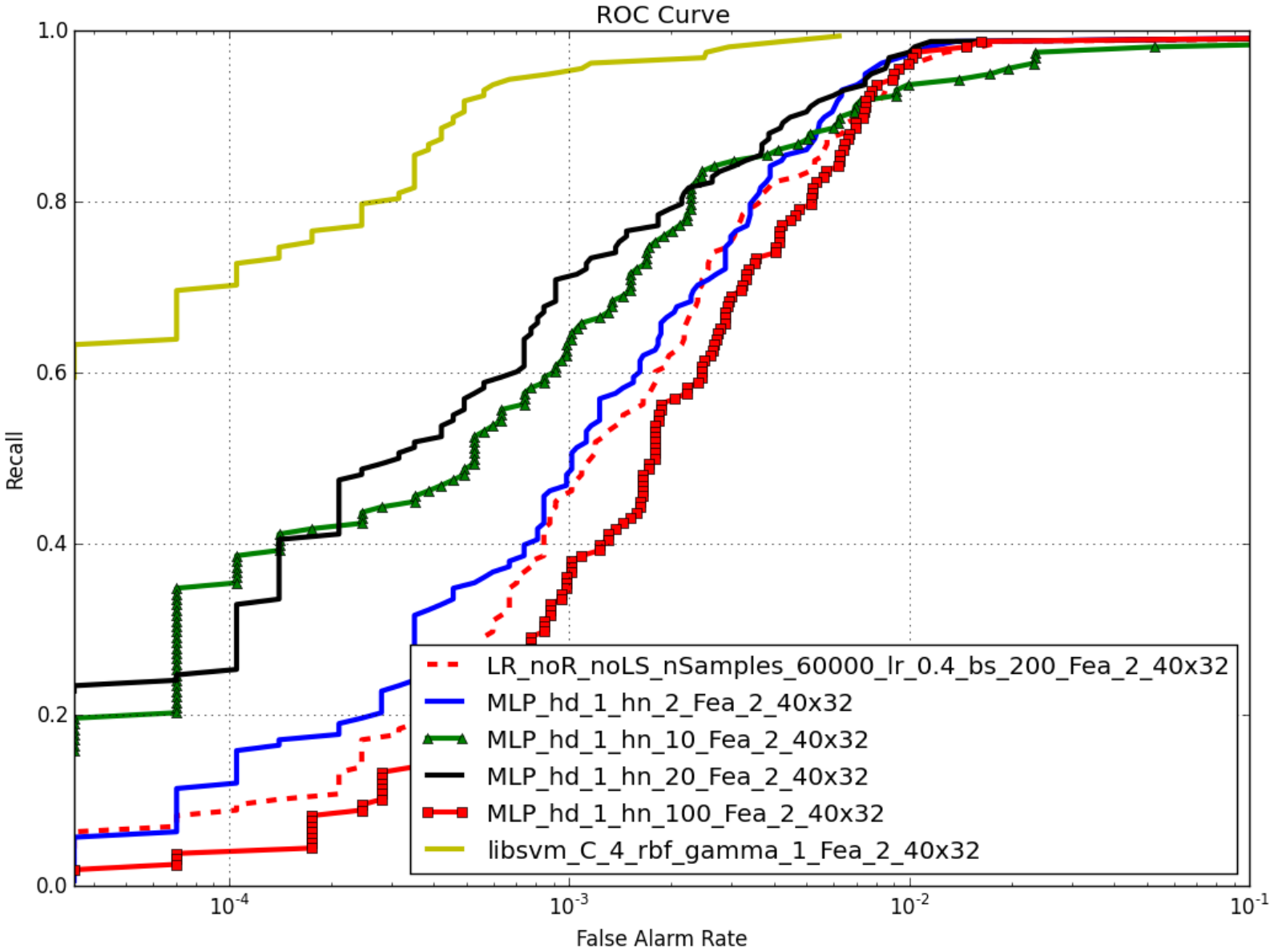
Basketball Dataset

- We have,
  - ✓ a training/validation set: more than 80,000 negatives and 465 positives for 2-frame samples, and more than 80,000 negatives and 586 positives for 1-frame samples. 80 percentage is treated as the training set and 20 percentage is put to validation set.
  - ✓ a testing set: 158 positives and 28451 negatives for 2-frame samples and similar number of samples for the 1-frame case.

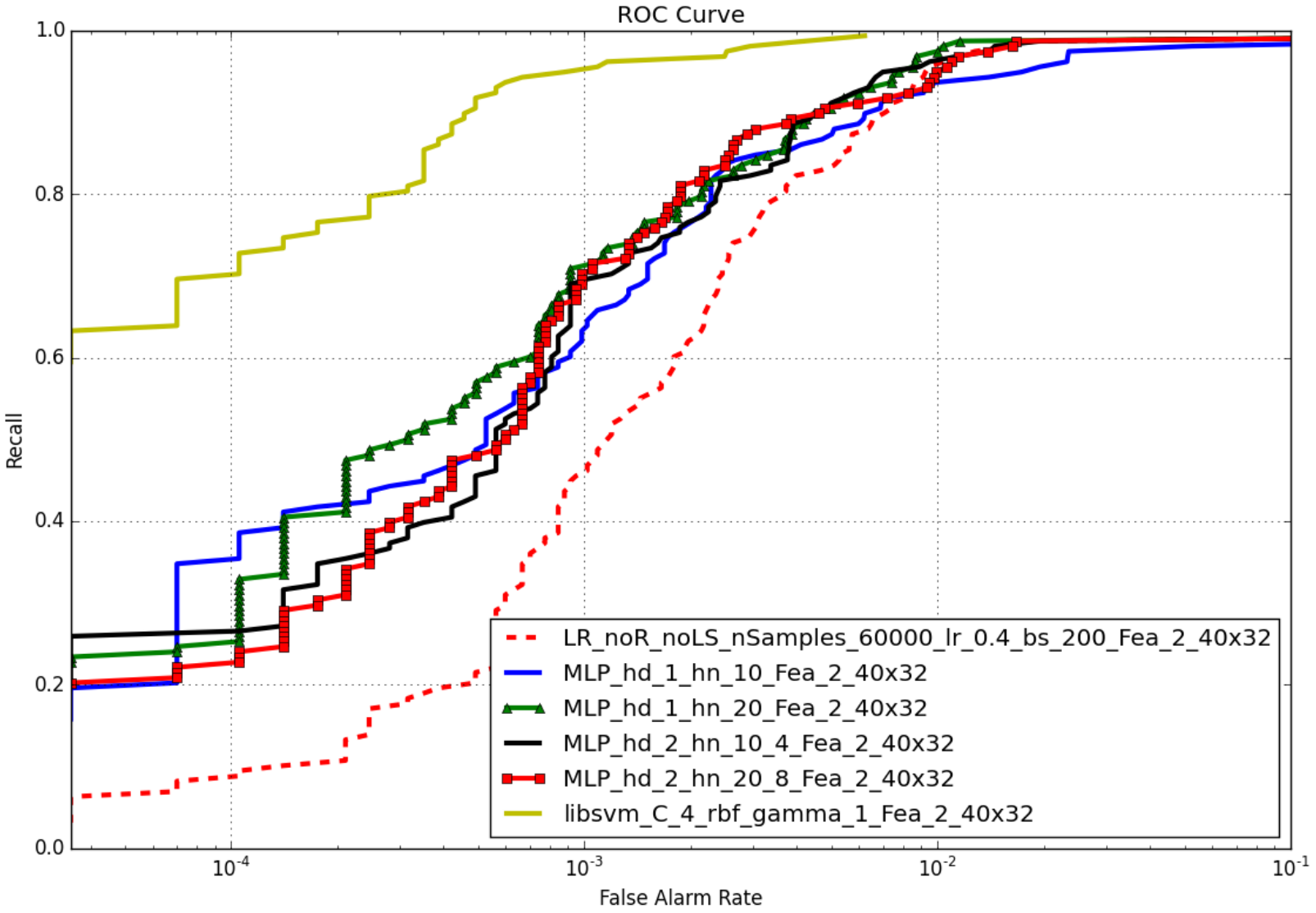
Results on basketball dataset

- Number of hidden layers: ?, number of nodes in hidden layer: ?
- Activation function in the hidden layer: tanh:
- Output layer: logistic regression (softmax)
- Loss function: negative Log-likelihood

#hidden layers: 1, #nodes in hidden layer: 2, 10, 20, 100



Number of hidden layers: 2, number of nodes in hidden layer: (20, 8), (10, 4)



References

- New UFLDL tutorial: http://deeplearning.stanford.edu/tutorial/
- LISA Deep learning tutorial: http://deeplearning.net/tutorial/mlp.html
- Yann Lecun et al., Efficien BackProp, http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf
- ICDAR'03: Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis. (How to deal with small number of samples)

Assignments

- Derive every formula by yourself in the slides
- Read the MLP code on deep learning tutorial, run it on MNIST dataset, and rewrite / change it to be theano-free.
- Carry out all experiments I showed on the basketball dataset using multi-lay neural network

Next lecture: Deep Convolutional Neural Network